

C# Avançado

Programação Multithread




Softblue
cursos online

Tópicos Abordados

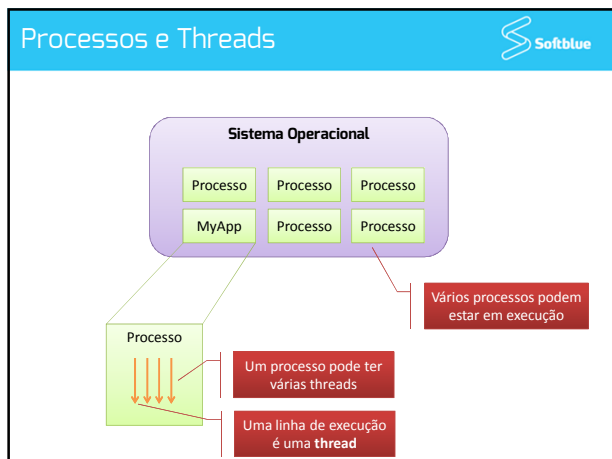


- Processos e threads
- Threads em C#
 - Prioridades em threads
 - Foreground e background threads
- Sincronismo
 - *lock* e *Monitor*
 - Atributo *Synchronization*
 - *Interlocked*
 - *Mutex*
 - *Semaphore*

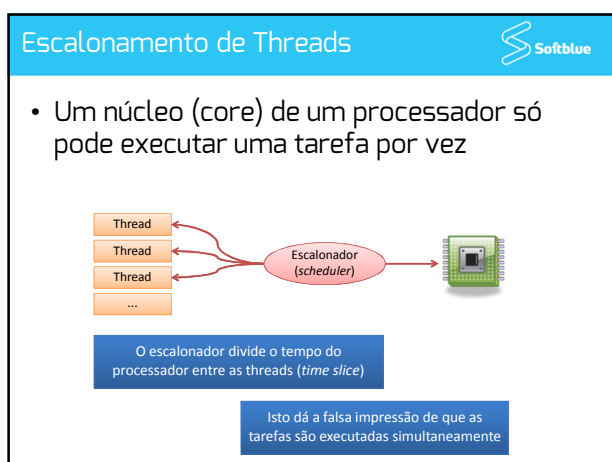
Tópicos Abordados



- Sincronismo
 - Events
 - *AutoResetEvent*
 - *ManualResetEvent*
 - *CountdownEvent*
 - *Wait()* e *Pulse()*
- Timers
 - *System.Threading.Timer*
 - *System.Timers.Timer*



- ## Threads
- Threads são consideradas processos leves
 - Um processo pode ter uma ou mais threads em execução "simultânea"
 - As threads de um processo compartilham o heap do processo
 - Área de memória onde ficam armazenados os objetos
 - Muitas aplicações são multithread
 - Ex: editor de texto



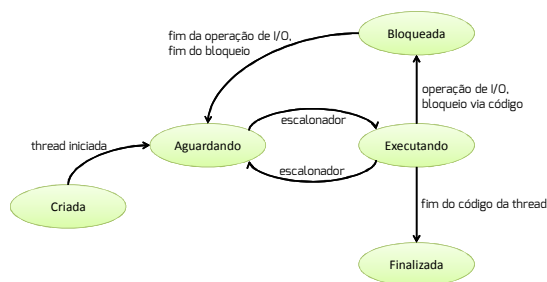
Escalonamento de Threads



- Na presença de múltiplos processadores ou processadores multi-core, é possível a execução verdadeiramente simultânea



Estados de Uma Thread



Criando Threads em C#



- A classe *Thread* é a forma mais básica de criar threads em aplicações C#
– Namespace *System.Threading*

```
static void Main()  
{  
    Thread t = new Thread(new ThreadStart(Run));  
    t.Start();  
    //...  
}  
  
static void Run()  
{  
    //...  
}
```

Delegate que define o método a ser executado pela nova thread

O método **Start()** inicia a thread

Quando o método **Run()** termina, a thread termina

Depois da chamada ao método *Start()*, a thread principal continua a execução do método *Main()*, enquanto a nova thread executa o método *Run()*

Criando Threads em C#

Softblue

- A criação do delegate pode ser substituída apenas pelo nome do método

```
Thread t = new Thread(Run);
```

- O delegate *ParameterizedThreadStart* permite passar um parâmetro para a thread

```
Thread t = new Thread(new ParameterizedThreadStart(Run));
t.Start("abc");
```

```
static void Run(object param)
{
    string str = (string)param;
}
```

Apenas um parâmetro do tipo *object* é permitido

Neste caso, o delegate também pode ser substituído pelo nome do método

O delegate também pode ser substituído por uma expressão lambda

Atuando na Thread Corrente

Softblue

- A property estática *CurrentThread* retorna o objeto *Thread* relativo à thread que está executando

```
Thread t = Thread.CurrentThread;
t.Name = "New Thread";
```

Dá uma nome para a thread

- O método estático *Sleep()* faz a thread corrente ficar bloqueada por um tempo

```
Thread.Sleep(2000);
```

Para por 2 segundos

Depois dos 2 segundos, a thread volta para o estado *Aguardando*

Prioridade em Threads

Softblue

- Threads têm uma prioridade associada
- Definidas no enum *ThreadPriority*
 - *Lowest*, *BelowNormal*, *Normal*, *AboveNormal*, *Highest*
- Por padrão, toda thread tem prioridade *Normal*
- A property *Priority* é usada para mudar a prioridade

```
t.Priority = ThreadPriority.AboveNormal;
```

Considerações Sobre Prioridades



- A prioridade é apenas uma dica para o escalonador
- Mesmo que uma thread tenha prioridade *Highest*, ainda assim outras podem executar antes dela
- Alterar a prioridade de threads pode fazer com que outras threads deixem de executar de forma adequada
- Na prática, dificilmente será necessário mudar a prioridade de uma thread
- Se for realmente necessário, faça isto com cuidado

Foreground e Background Threads



- A property *IsBackground* define o tipo
- ```
t.IsBackground = true;
```
- Background thread
- Quando o método *Main()* termina e apenas *background threads* ainda estão executando, elas são abortadas e a aplicação termina
  - Quando o método *Main()* termina e existe alguma *foreground thread* executando, a aplicação só termina quando a thread terminar de executar

---

---

---

---

---

---

---

## Outros Métodos da Classe *Thread*



| Método             | Descrição                                                           |
|--------------------|---------------------------------------------------------------------|
| <i>Abort()</i>     | Para uma thread bloqueada, fazendo com que uma exceção seja lançada |
| <i>Interrupt()</i> | Funciona de forma semelhante ao <i>Abort()</i>                      |
| <i>Suspend()</i>   | Suspende a execução da thread                                       |
| <i>Resume()</i>    | Retoma a execução de uma thread suspensa                            |

*Abort()* e *Interrupted()* são perigosos

*Suspend()* e *Resume()* são obsoletos

Não use nenhum destes métodos

---

---

---

---

---

---

---

## O Método `Join()`



- Faz com que a thread aguarde até que outra thread finalize a sua tarefa

```
Thread t = new Thread(Run);
t.Start();

//...
t.Join();

//...
```

Bloqueia a thread em execução até que `t` termine de executar

A execução só continua depois que `t` parou de executar

O `Join()` é útil em situações onde a continuação do código depende do resultado de outras threads

---

---

---

---

---

---

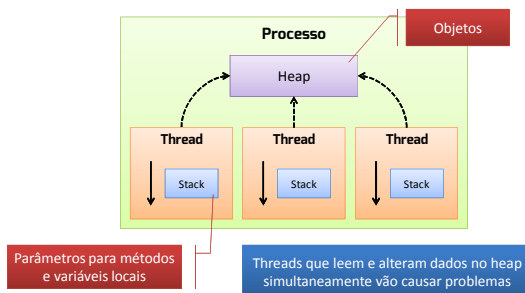
---

---

## Compartilhamento de Dados



- Threads compartilham o heap do processo



---

---

---

---

---

---

---

---

## Sincronismo



- Difícilmente uma aplicação vai executar threads totalmente independentes umas das outras
- Normalmente é necessário que haja uma sincronização nessa execução, bem como comunicação entre as threads
- C# tem uma série de recursos que visam atender a necessidade de sincronismo em aplicações multithread

---

---

---

---

---

---

---

---

## Sincronismo com *lock*



- Define um bloco de código que não pode ser executado simultaneamente por mais de uma thread
- Usado para evitar o acesso simultâneo a dados compartilhados, evitando as *race conditions*
- É preciso definir um objeto que fornece um token de acesso (um lock)
- Uma thread só consegue executar o bloco se estiver de posse do lock

---

---

---

---

---

---

---

## Sincronismo com *lock*



```
class Counter
{
 public int Count { get; private set; }
 public void Increment() { Count++; }
}
```

```
private object sync = new object();
```

```
public void Execute()
```

```
{
```

```
 Counter c = new Counter();
```

```
 for (int i = 0; i < 5; i++)
```

```
 {
```

```
 new Thread(() => Run(c)).Start();
```

```
 }
```

```
void Run(Counter c)
```

```
{
```

```
 lock (sync)
```

```
 {
```

```
 c.Increment();
```

```
 Console.WriteLine(c.Count);
```

```
 }
```

```
}
```

Objeto que fornece o lock

As 5 threads compartilham o mesmo objeto Counter

O que está definido no bloco lock só pode ser executado pela thread que possui o lock

---

---

---

---

---

---

---

## Usando a Classe *Monitor*



- A classe *Monitor* permite fazer um sincronismo semelhante ao do *lock*
- Durante o processo de compilação, o *lock* é transformado em chamadas a métodos da classe *Monitor*
- Usar a classe *Monitor* diretamente permite ter mais controle sobre o sincronismo

---

---

---


---

---

---

---

## Usando a Classe *Monitor*



```
private object sync = new object();
void Run(Counter c)
{
 bool lockTaken = false;
 try
 {
 Monitor.Enter(sync, ref lockTaken);
 c.Increment();
 Console.WriteLine(c.Count);
 }
 finally
 {
 if (lockTaken)
 {
 Monitor.Exit(sync);
 }
 }
}
```

**Monitor.Enter()** obtém o lock

**Monitor.Exit()** libera o lock

Outras recursos presentes na classe *Monitor* serão abordados na sequência

---

---

---

---


---

---

---

---

## Reentrant Locks



- Uma thread pode obter o lock do mesmo objeto repetidamente

```
lock (sync)
{
 //...
 lock (sync)
 {
 //...
 }
}
```

- O lock só é liberado quando o último bloco *lock* é finalizado

---

---

---

---


---

---

---

---

## Objeto Usado para o Sincronismo



- A forma mais comum é declarar um objeto interno para fornecer o lock

```
private object sync = new object();
```

- No sincronismo de elementos estáticos, é preciso usar um objeto estático

```
private static object sync = new object();
```

---

---

---

---

---

---

---

---



## Objeto Usado para o Sincronismo



- O lock da própria instância também pode ser utilizado

```
lock(this)
{
 //...
}
```

- Para elementos estáticos, o lock pode ser obtido a partir do tipo

```
lock(typeof(MyClass))
{
 //...
}
```

---

---

---

---

---

---

---

## Atributo *Synchronization*



- Uma classe que possui o atributo *Synchronization* não permite acesso concorrente a nenhum dos elementos dos objetos da classe
  - Namespace *System.Runtime.Remoting.Contexts*
- A classe deve herdar de *ContextBoundObject*

```
[Synchronization]
class MyClass : ContextBoundObject
{
 //...
}
```

Tudo no objeto é sincronizado, mesmo o que não é necessário

Use este recurso com cuidado, pois pode degradar a performance

---

---

---

---

---

---

---

## A Classe *Interlocked*



- Possui métodos estáticos para sincronizar operações simples, como atribuição de valor, incremento, decremento, etc.

```
void Method()
{
 x++;
}
```

```
void Method()
{
 Interlocked.Increment(ref x);
}
```

```
void Method()
{
 x = 15;
}
```

```
void Method()
{
 Interlocked.Exchange(ref x, 15);
}
```

```
void Method()
{
 if (x == 5)
 {
 x = 10;
 }
}
```

```
void Method()
{
 Interlocked.CompareExchange(ref x, 10, 5);
}
```

---

---

---

---

---

---

---

## A Classe *Mutex*



- Funciona como o *lock*
  - Apenas uma thread executa por vez
- A grande diferença é que o *Mutex* pode ser usado entre processos diferentes

```
Mutex mutex = new Mutex(false, "MyMutex");
if (mutex.WaitOne())
{
 try
 {
 //...
 }
 finally
 {
 mutex.ReleaseMutex();
 }
}
```

Cria o *Mutex* com um nome

Obtém o lock

Libera o lock

Um *Mutex* pode ser usado para evitar que a mesma aplicação seja executada mais de uma vez

## As Classes *Semaphore* e *SemaphoreSlim*



- Permitem que mais de uma thread executem o código simultaneamente, mas estipulam um limite
- Diferença
  - *Semaphore* permite o sincronismo entre processos
  - *SemaphoreSlim* não permite

```
SemaphoreSlim semaphore = new SemaphoreSlim(3);
semaphore.Wait();
//...
semaphore.Release();
```

O construtor informa quantas threads podem executar simultaneamente

*Wait()* obtém o lock

*Release()* libera o lock

## Sincronização com Events



- Não tem relação com a palavra-chave *event*
- Executa o sincronismo baseado em *signals*
  - Uma thread bloqueia até receber um aviso indicando que pode continuar a executar
- Classes
  - *AutoResetEvent*
  - *ManualResetEvent*
  - *ManualResetEventSlim*
  - *CountdownEvent*

## AutoResetEvent



- Funciona como uma cancela
  - Quando a cancela é aberta, apenas uma thread executa
  - Assim que a thread começa a executar, a cancela fecha
- Herda de *EventWaitHandle*
  - `AutoResetEvent h = new AutoResetEvent(false);`
- Métodos
  - *WaitOne()*
    - Passa pela cancela se ela estiver aberta. Senão bloqueia.
  - *Set()*
    - Abre a cancela para uma thread passar
  - *Reset()*
    - Fecha a cancela

Indica se a "cancela" está inicialmente aberta

## ManualResetEvent / ManualResetEventSlim



- Funcionam como o *AutoResetEvent*, mas permitem a passagem de múltiplas threads
  - A cancela deve ser fechada explicitamente
- *ManualResetEventSlim* não permite a sincronização entre processos
- *ManualResetEvent* herda de *EventWaitHandle*

```
ManualResetEventSlim h = new ManualResetEventSlim(false);
```

- Métodos
  - *WaitOne()* / *Wait()*
  - *Set()*
  - *Reset()*

## CountdownEvent



- Permite que uma thread espere mais de uma thread executar
- A thread bloqueada tem um contador e só executa quando ele chegar no valor 0

```
CountdownEvent h = new CountdownEvent(3);
```

Número de threads que serão aguardadas

- Métodos
  - *Wait()*
    - Bloqueia até o contador chegar em 0
  - *Signal()*
    - Sinaliza, decrementando o contador
  - *Reset()*
    - Restaura o valor original do contador

## Sincronismo com *Wait()* e *Pulse()*



- A classe *Monitor* possui três métodos estáticos usados para sincronismo
  - *Wait()*
    - Bloqueia a thread
  - *Pulse()*
    - "Acorda" uma thread bloqueada
  - *PulseAll()*
    - "Acorda" todas as threads bloqueadas
- Estes métodos devem ser chamados dentro de um contexto de lock
  - Bloco *lock* ou *Monitor.Enter()*/*Monitor.Exit()*
- O método *Wait()* libera o lock da thread

---

---

---

---

---

---

---

## Timers



- Permitem executar um método de forma repetida, de acordo com um intervalo de tempo definido
- Dois timers multithread disponíveis
  - *System.Threading.Timer*
  - *System.Timers.Timer*
- O método é executado em uma thread a parte
  - Thread obtida do pool de threads

---

---

---

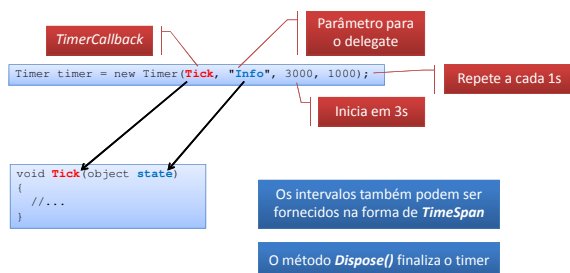
---

---

---

---

## *System.Threading.Timer*



---

---

---

---

---

---

---

- Internamente, funciona como um *System.Threading.Timer*
- Possui algumas diferenças no uso

```
Timer timer = new Timer();
timer.Interval = 1000;
timer.Elapsed += Tick;
timer.Start();
```

O método **Stop()** para o timer

O método **Dispose()** finaliza o timer

```
void Tick(object sender, EventArgs args)
{
 //...
}
```

---

---

---

---

---

---

---



---

---

---

---

---

---

---